# CORE JAVA MODULE 2 NOTES :

26 June 2025 12:57

## ➢ Module 2: Variables, Operators, Data Types, Input

| No. | Topic |
|---|---|
| 2.1 | Variables and Data Types (Primitive & Non-Primitive) |
| 2.2 | Type Conversion and Type Casting |
| 2.3 | Arithmetic, Logical, Relational, Assignment Operators |
| 2.4 | Unary, Ternary, Operators |
| 2.5 | Taking Input using Scanner Class |
| 2.6 | Wrapper Classes & Autoboxing / Unboxing |
| 2.7 | Interview Questions and Assignment |

## ⇒ Module-2.1 : Real-World Analogy: Variables and Data Types

### ✧ Imagine: You're managing a kitchen shelf.

Each **container (dabba)** on the shelf holds something:

- One has **sugar**
- One has **rice**
- One has **salt**
- One has a **label**: "Chilli Powder"
- One is **empty**, but ready to be used

Now:

- Every container holds **one type of item**
- You **label** each one (so you remember what's inside)
- You can **change the content**, but the container type stays the same

### Mapping to Java:

| Kitchen Concept | Java Concept |
|---|---|
| ◆ Container (dabba) | ◆ Variable |
| ◆ What's inside | ◆ Value |
| ◆ Label on container | ◆ Variable name |
| ◆ Type of item allowed | ◆ Data type (e.g., int, float) |
| ◆ Empty container | ◆ Declared but uninitialized variable |

### ✧ What is a Variable?

A **variable is just like a container or it** is a name given to a memory location that stores a value of a specific data type.

Example :

- sugar = 5;
- nameTag = "TATA Salt";

### ✧ NAMING CONVENTION:

```
//  1. Variable names should use **camelCase**
int studentAge = 18;
String studentName = "Jatin";

//  2. Use meaningful, descriptive names
int totalMarks = 450;          // Good
int x = 450;                   // Bad  (not meaningful)
```

```
//  3. Start with a **letter**, **not number or symbol**
int _temp = 30;                  // Allowed but avoid starting with _ or $
// int 2ndRank = 2;                Not allowed (starts with number)

//  4. No spaces or special characters
// int total marks = 90;         Not allowed (space)
// int total@marks = 90;         Not allowed (special character)

//  5. Java is **case-sensitive**
int number = 10;
int Number = 20;                 // Both are different variables

//  6. Avoid Java reserved keywords
// int class = 10;               Not allowed (class is a keyword)

//  7. Constants should be in **ALL_CAPS**
final int MAX_SCORE = 100;
```

## ✧ What is a Data Type in Java?

A **data type** tells Java **what kind of data** a variable will store and **how much memory** to reserve for it.

**Simple Definition:**

It's like **telling Java**:
"Hey, this variable will store a number!" or "This will store text!" or "This will be true/false!"

Code : Demo:

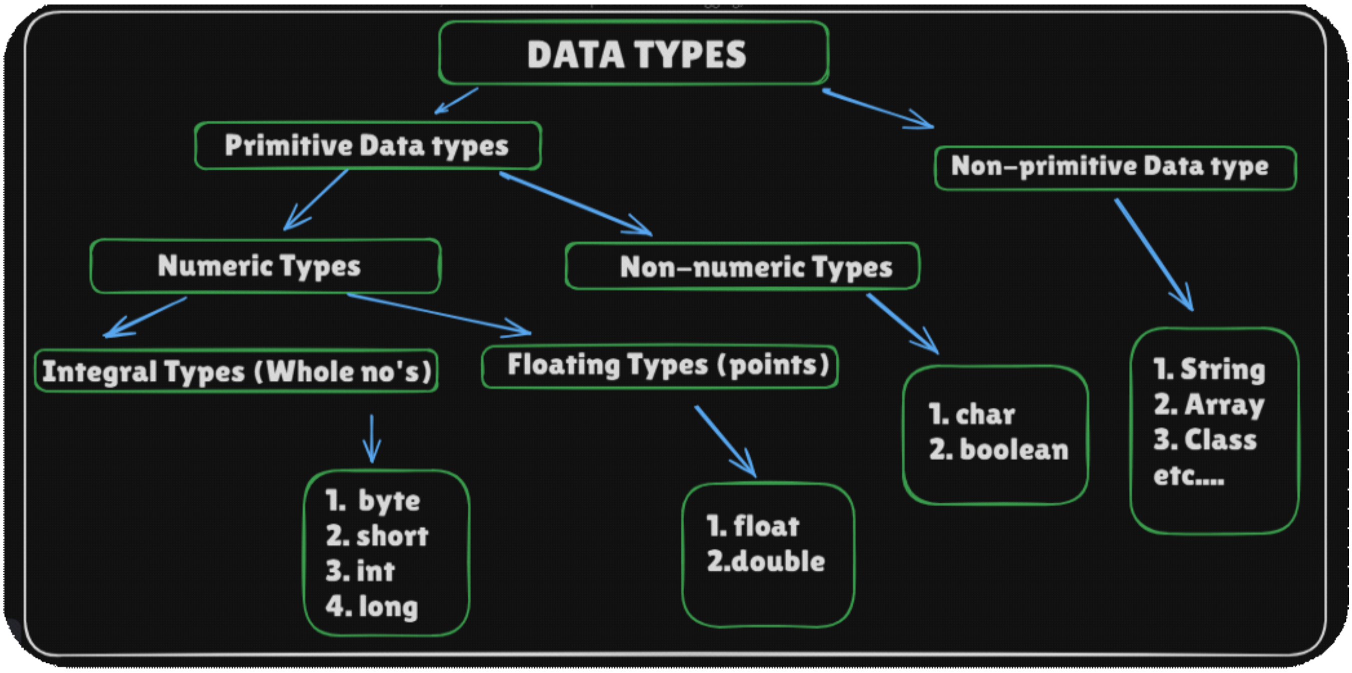
```
public class KitchenShelf {
   public static void main(String[] args) {
      int riceKg = 5;
      float oilLitres = 1.5f;
      char spiceGrade = 'A';
      boolean isFresh = true;
      String brand = "TATA Salt";

      System.out.println("Rice: " + riceKg + "kg");
      System.out.println("Oil: " + oilLitres + "L");
      System.out.println("Grade: " + spiceGrade);
      System.out.println("Fresh? " + isFresh);
      System.out.println("Brand: " + brand);

     // To find the max value of each data type : we have to use each datatype wrapper class:

     System.out.println("Max byte     : " + Byte.MAX_VALUE);
     System.out.println("Max short    : " + Short.MAX_VALUE);
     System.out.println("Max int      : " + Integer.MAX_VALUE);
     System.out.println("Max long     : " + Long.MAX_VALUE);
     System.out.println("Max float    : " + Float.MAX_VALUE);
     System.out.println("Max double   : " + Double.MAX_VALUE);
     System.out.println("Max char     : " + (int) Character.MAX_VALUE); // Unicode value
     System.out.println("Boolean has only: true/false");
   }
}
```

## ✧ Java Data Types Are of 2 Main Types:



## ✧ Descriptions of Each Primitive Data Type (8 Total)

### 1 byte

- **Size**: 1 byte (8 bits)
- **Range**: –128 to +127
- **Stores**: Very small whole numbers
- **Use when**: You need to store tiny values and want to save memory
- **Example**: Baby age in months (12), LED brightness level (0–100)

### 2 short

- **Size**: 2 bytes (16 bits)
- **Range**: –32,768 to +32,767
- **Stores**: Small whole numbers
- **Use when**: Values are bigger than byte but still not huge
- **Example**: Height in cm (170), PIN codes, year
- **Tip**: Rarely used; only if you want to save memory compared to int

### 3 int (Default whole number type)

- **Size**: 4 bytes (32 bits)
- **Range**: –2,14,74,83,648 to +2,14,74,83,647
- **Stores**: Normal whole numbers
- **Use when**: You want to store counts, roll numbers, salary, IDs
- **Example**: Student roll number = 101, Salary = 50000
- **Tip**: Most commonly used integer type

### 4 long

- **Size**: 8 bytes (64 bits)
- **Range**: ±9,22,33,72,03,68,54,77,580 (very big)
- **Stores**: Very large whole numbers
- **Use when**: int is not enough
- **Example**: Mobile number, Population count, System time
- **Tip**: Use L at the end → long phone = 9876543210L;

### 5 float

- **Size**: 4 bytes
- **Precision**: ~6 to 7 decimal digits
- **Stores**: Decimal numbers (less accurate)
- **Use when**: You need decimal numbers but want to save memory

- **Example**: Weight = 52.5f, Temperature = 36.7f
- **Tip**: Must add f at the end → float weight = 55.5f;

### 6 double (Default decimal type)

- **Size**: 8 bytes
- **Precision**: ~15 decimal digits
- **Stores**: Decimal numbers (more accurate)
- **Use when**: You need accuracy like money or scientific data
- **Example**: Bank balance = 99999.99, Pi = 3.141592653589
- **Tip**: Use this for most decimal operations

### 7 char

- **Size**: 2 bytes
- **Range**: One Unicode character (like A–Z, 0–9, ₹, )
- **Stores**: A single character
- **Use when**: You need to store only one letter or symbol
- **Example**: Grade = 'A', Symbol = '₹', Key = 'Y'
- **Tip**: Use single quotes → char grade = 'A';

### 8 boolean

- **Size**: 1 bit (internally stored as 1 byte)
- **Values**: true or false
- **Stores**: Yes/No, On/Off, True/False
- **Use when**: You want to store a condition or decision
- **Example**: isLogin = true;, isFresh = false;
- **Tip**: Perfect for if conditions and flags

## ✧ Difference Between Primitive and Non-Primitive Data Types

| Feature | Primitive Data Types | Non-Primitive Data Types |
|---|---|---|
| 1. **Definition** | Basic built-in data types provided by Java | Derived or created using classes/objects |
| 2. **Total Types** | Only 8 (int, char, boolean, etc.) | Many (String, Array, Class, Object, etc.) |
| 3. **Stores** | Actual value directly in memory | Reference (address) to object in heap |
| 4. **Memory Location** | Stored in **stack memory** | Reference stored in stack, data stored in **heap** |
| 5. **Operations** | Can do direct operations (+, –, *, /) | Need methods/functions to manipulate |
| 6. **Default Values** | Simple default values (e.g., 0, false) | null (means no object assigned yet) |
| 7. **Null Allowed?** | Cannot be null | Can be null |
| 8. **Size is Fixed?** | Yes – depends on data type | No – depends on object (like String length, array size) |
| 9. **Examples** | int, float, char, boolean, byte, etc. | String, Array, Class, Interface, Object |
| 10. **Created by?** | Java Language (predefined) | Created using **classes or APIs** |

## ⇒ Module 2.2: Type Casting (Type Conversion of Primitives)

### ◈ What is Type Casting in Java?

**Type Casting** means **changing the data type** of a value from one **primitive type** to another.

## ✧ Two Types of Type Casting in Java:

**1. Implicit Type Casting (Widening Conversion)**

- Automatically done by Java — safe and no data loss
  **Small type → Bigger type**

**Example:**

```
int a = 10;
double b = a;  // int → double
System.out.println(b);  // Output: 10.0
```

*Real-life:* Pouring a glass of water into a bucket — nothing spills

**2. Explicit Type Casting (Narrowing Conversion)**

- Done manually by the programmer with the help of cast operator (datatype):— may cause data loss
  **Bigger type → Smaller type**

**Example:**

```
double x = 9.75;
int y = (int) x;  // double → int
System.out.println(y);  // Output: 9
```

*Real-life:* Pouring a bucket of water into a glass — it may overflow (data loss)

## ✧ if explicit type casting can cause data loss... then why should we even use it?

**We use explicit type casting when we're sure that the value will fit safely into the smaller type** — and we want to either:

1️⃣ **Remove the decimal part on purpose**

```
double price = 99.99;
int rounded = (int)price;  // Output: 99
```

*Useful when you want only the whole number (e.g., rounded scores, ticket counts).*

2️⃣ **Save memory**

If you know your value is small (e.g., 0–100), why use int (4 bytes) when byte (1 byte) is enough?

```
byte level = (byte)20;
```

*Used in games, embedded systems, or apps with limited memory.*

3️⃣ **Match method requirements**

Sometimes you use libraries or code where the method only accepts a smaller type like byte or short — so you cast to fit it.

```
sendData((byte)1);  // method expects byte
```

**REMEMBER:**

Only use explicit casting when **you are 100% sure** that the value fits.
If not, it may lead to **wrong output or data loss**.

**Line to remember:**

We don't use it blindly — we use it **when we need control** and **know the risks**. That's what makes us smart programmers!

## NOTE:

💎 What we learned today — Type Casting — applies only to **primitive types** like int, float, double, char, etc.
**But in future**, when we learn about **String, Arrays, and Wrapper classes**, we will also learn how to:

- Convert "123" (String) → 123 (int)
- Convert 10 (int) → "10" (String)

This is called **Type Conversion between non-primitive types**, and it's done using methods like parseInt() and valueOf().
We will cover that **when we study String and Wrapper classes** in upcoming modules.

## Module: 2.3 : What is an Operator in Java?

**An operator is a special symbol** used to perform **operations** on variables and values.

**In simple words**:
An operator **tells Java what to do** with the data.

## Types of Operators in Java

| No. | Type of Operator | Purpose | Example |
|---|---|---|---|
| 1 | **Arithmetic Operators** | Perform basic math | + - * / % |
| 2 | **Relational Operators** | Compare values (true/false) | > < == != |
| 3 | **Logical Operators** | Combine multiple conditions | `&& |
| 4 | **Assignment Operators** | Assign values to variables | = += -= |
| 5 | **Unary Operators** | Work with a single operand | ++ -- + - ! |
| 6 | **Bitwise Operators** | Work at the binary level | `& |
| 7 | **Ternary Operator** | Shortcut for if-else | ? : |
| 8 | **Shift Operators** | Shift bits left or right | << >> >>> |

## 1. Arithmetic Operators

Used for **mathematical operations**.

| Operator | Meaning | Example | Output |
|---|---|---|---|
| + | Addition | 5 + 3 | 8 |
| - | Subtraction | 5 - 3 | 2 |
| * | Multiplication | 5 * 3 | 15 |
| / | Division | 6 / 3 | 2 |
| % | Modulus (remainder) | 5 % 3 | 2 |

**Real-Life Analogy**:
Think of it like a calculator for your code. Want to split ₹100 between 4 friends? Use /. Want the leftover chocolate after dividing? Use %.

## 2. Relational Operators (Comparison Operators)

Used to **compare two values** and return true or false.

| Operator | Meaning | Example | Output |
|---|---|---|---|
| == | Equal to | 5 == 5 | true |
| != | Not equal to | 5 != 3 | true |
| > | Greater than | 5 > 3 | true |
| < | Less than | 5 < 3 | false |
| >= | Greater than or equal | 5 >= 5 | true |
| <= | Less than or equal | 5 <= 2 | false |

**Real-Life Analogy**:
Just like comparing marks of two students — "Is A's marks > B's marks?"

## 3. Logical Operators

Used to **combine multiple conditions**.

| Operator | Meaning | Example | Output |
|---|---|---|---|
| && | Logical AND | true && true → true | true |
| \|\| | Logical OR | True\|\|false → true | true |
| ! | Logical NOT (negation) | !true → false | false |

**Real-Life Analogy**:

- Want to go for a trip? You need: money && permission
- Okay with any one? Then: money || permission

- Not a no-ball? Then count it: !isNoBall

## 4. Assignment Operators

Used to **assign values** to variables.

| **Operator** | **Meaning** | **Example** | **Result** |
|---|---|---|---|
| = | Assign | a = 5 | a = 5 |
| += | Add and assign | a += 3 (a = a + 3) | a = 8 |
| -= | Subtract and assign | a -= 2 | a = 6 |
| *= | Multiply and assign | a *= 2 | a = 12 |
| /= | Divide and assign | a /= 2 | a = 6 |
| %= | Modulus and assign | a %= 5 | a = 1 |

**Real-Life Analogy**:
Think of = as "give this value to this variable". += is like adding something to your bank account.

## 5. What are Unary Operators in Java?

A **Unary Operator** is an operator that **works with only one operand**.
In short: it does something to **just one value** — like increase it, decrease it, or reverse it.

### ✧ Types of Unary Operators in Java

| Operator | Name | Description |
|---|---|---|
| + | Unary Plus | Indicates a positive value (rarely used) |
| - | Unary Minus | Negates a number (makes positive → negative) |
| ++ | Increment | Increases value by 1 |
| -- | Decrement | Decreases value by 1 |
| ! | Logical NOT | Reverses boolean value |

### ✧ Each with Examples and Real-Life Meaning

#### 1️⃣ Unary Plus +a

Shows that the number is positive.

```
int a = +10;  // same as just int a = 10;
```

**Use case**: Rare — usually numbers are positive by default.

#### 2️⃣ Unary Minus -a

Converts positive to negative and vice versa.

```
int a = 5;
int b = -a;   // b = -5
```

**Analogy**: Flip the sign — like flipping a coin from head to tail.

#### 3️⃣ Increment ++

Increases the value by 1

◈ **Pre-Increment (++a)**

***First increment then print***

```
int a = 5;
int result = ++a;  // a = 6, result = 6
```

◈ **Post-Increment (a++)**
***First print then increment***

```
int a = 5;
int result = a++;  // result = 5, then a = 6
```

## 4 Decrement --

Decreases the value by 1
◈ **Pre-Decrement (--a)**
***First decrement then print***

```
int a = 5;
int result = --a;  // a = 4, result = 4
```

◈ **Post-Decrement (a--)**
***First print then decrement***

```
int a = 5;
int result = a--;  // result = 5, then a = 4
```

## 5 Logical NOT !

Reverses a boolean value

```
boolean isRaining = false;
System.out.println(!isRaining);  // true
```

**Analogy**: "NOT Raining? Then let's go outside!"
!false → true

## 6. What is the Ternary Operator?

The **Ternary Operator (?:)** is a shortcut for if-else in Java.
It's the **only operator in Java that takes 3 operands** — that's why it's called **ternary**.

### Syntax:

```
condition ? value_if_true : value_if_false;
```

### Read it like:

"If condition is true, then do this, otherwise do that."

## Example:

```
int age = 20;
String result = (age >= 18) ? "Adult" : "Minor";
System.out.println(result);  // Output: Adult
```

## Real-Life Analogy:

Suppose your **marks** decide whether you **pass or fail**:

```
int marks = 40;
String result = (marks >= 33) ? "Pass" : "Fail";
```

## Ternary Operator vs If-Else

### Using if-else:

```
if (marks >= 33) {
   result = "Pass";
} else {
   result = "Fail";
}
```

**Using ternary (shorter & cleaner):**

result = (marks >= 33) ? "Pass" : "Fail";

## ⇒ Module-2.4 : Taking Input using Scanner Class

### Real-Life :

- Imagine a **teacher** (Scanner) asking a student to write answers (inputs).
- The scanner **reads the answers** and stores them in your variables.

### What is the Scanner Class in Java?

Scanner is a built-in **Java class** used to **take input from the user**.
It is found in the package:

import java.util.Scanner;

### Syntax to Use:

Scanner sc = new Scanner(System.in);

Think of it like:
"Create a scanner object (sc) to read input from the **keyboard** (System.in)"

### ⇒ Steps to Take User Input:

1. Import the Scanner Class: To use the Scanner class, you must first import it:
2. Create a Scanner Object: Create an object of the Scanner class to read input from the standard input stream (keyboard):
   Ex : Scanner sc = new Scanner(System.in);

**Use Methods of Scanner Class: The Scanner class provides various methods to read different types of data:**

- nextInt() for integers
- hasNextInt(): Checks if the next token is an integer.
- nextDouble() for floating-point numbers
- hasNextDouble(): Checks if the next token is a double.
- nextLine() for entire lines of text
- next() for a single word or token
- nextBoolean() for boolean values

### CODE :

```
import java.util.Scanner;

public class SimpleScannerInput {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // 1. String (single word)
        System.out.print("Enter your first name (single word): ");
        String firstName = sc.next();  // reads up to first space
        sc.nextLine(); // to clear the buffer

        // 2. String (full line)
        System.out.print("Enter your full address: ");
        String address = sc.nextLine();  // reads entire line

        // 3. int
        System.out.print("Enter your age (int): ");
        int age = sc.nextInt();

        // 4. float
        System.out.print("Enter your marks (float): ");
        float marks = sc.nextFloat();

        // 5. double
        System.out.print("Enter your bank balance (double): ");
        double balance = sc.nextDouble();

        // 6. long
        System.out.print("Enter your phone number (long): ");
        long phone = sc.nextLong();

        // 7. boolean
        System.out.print("Are you vaccinated? (true/false): ");
        boolean vaccinated = sc.nextBoolean();

        // Print all values
        System.out.println("\n===== Your Info =====");
        System.out.println("Name       : " + firstName);
        System.out.println("Address    : " + address);
        System.out.println("Age        : " + age);
        System.out.println("Marks      : " + marks);
        System.out.println("Balance    : " + balance);
        System.out.println("Phone      : " + phone);
        System.out.println("Vaccinated : " + vaccinated);

        sc.close();  // always close the scanner
    }
}
```

## ⇒ Module-2.5 : Wrapper Classes & Autoboxing / Unboxing

### Problem with Primitive Data Types:

Java has **primitive types** like int, float, char, boolean etc.
Example:

int a = 10;
→ These are **not objects**, just simple data.

### But Java is Object-Oriented!

Java works best with **objects**, and many features like:

- Collections (ArrayList, HashMap, etc.)
- Generics (List<Integer>)
- Frameworks (Spring, Hibernate, etc.)

require **objects**, **not primitives**.

### Real Problem:

You **can't do this**:
ArrayList<int> list = new ArrayList<>();  //  Error: primitives not allowed

### But this works:

ArrayList<Integer> list = new ArrayList<>();

## ✧ So What Are Wrapper Classes?

**Wrapper classes** are object versions of primitive types.
They “wrap” the primitive value inside an object.

| Primitive | Wrapper Class |
|---|---|
| ◆ byte | ◆ Byte |
| ◆ short | ◆ Short |
| ◆ int | ◆ Integer |
| ◆ long | ◆ Long |
| ◆ float | ◆ Float |
| ◆ double | ◆ Double |
| ◆ char | ◆ Character |
| ◆ boolean | ◆ Boolean |

## ✧ When to Use

| Use Primitive | Use Wrapper Class |
|---|---|
| ◆ Simple calculations | ◆ Working with Collections |
| ◆ Low memory required | ◆ Generics / Frameworks |
| ◆ Performance needed | ◆ Null support needed |

## ✧ Behind the Scenes: Autoboxing & Unboxing

Java makes it **easy** to switch between primitive and wrapper automatically.

### ◈ Autoboxing:

Converting **primitive → object** (wrapper class)

```
int a = 10;
Integer obj = a;  // autoboxing happens here
Java does: Integer obj = Integer.valueOf(a);
```

### ◈ Unboxing:

Converting **object → primitive**

```
Integer obj = 20;
int b = obj;  // unboxing happens here
Java does: int b = obj.intValue();
```

## Analogy:

Imagine primitive types are like **raw food items**.
Wrapper classes are like **packaged versions** of those items — easy to deliver, store, and use in modern systems (like ArrayList or Java APIs).

## ➢ Interview Questions :

### Beginner-Level Questions & Answers

| No. | Question | Answer |
|---|---|---|
| 1 | ◆ What are primitive data types in Java? | ◆ Data types that store simple values directly (like int, float, char, boolean, etc.). |

| | | |
|---|---|---|
| 2 | ◆ What is a non-primitive (reference) data type? | ◆ Data types that store references to objects (like String, arrays, classes). |
| 3 | ◆ How many primitive data types are there in Java? | ◆ 8 (byte, short, int, long, float, double, char, boolean). |
| 4 | ◆ What is type casting? | ◆ Converting one data type into another manually (e.g., double to int). |
| 5 | ◆ What is the difference between = and ==? | ◆ = assigns values, == compares values. |
| 6 | ◆ What is ++a and a++? | ◆ ++a is pre-increment (increases first), a++ is post-increment (increases after). |
| 7 | ◆ What is a ternary operator? | ◆ A shortcut for if-else: condition ? true_value : false_value. |
| 8 | ◆ How to take string input with space using Scanner? | ◆ Use sc.nextLine() instead of sc.next(). |
| 9 | ◆ What is a wrapper class? | ◆ A class that wraps a primitive in an object (e.g., int → Integer). |
| 10 | ◆ Why do we need wrapper classes? | ◆ To use primitive values with collections, generics, and Java APIs. |
| 11 | ◆ What is autoboxing? | ◆ Automatically converting a primitive into a wrapper object. |
| 12 | ◆ What is unboxing? | ◆ Converting a wrapper object back into a primitive. |
| 13 | ◆ How to take input from the user in Java? | ◆ Use the Scanner class from java.util package. |
| 14 | ◆ What does next() do in Scanner? | ◆ Reads a single word (until space). |
| 15 | ◆ What does nextLine() do in Scanner? | ◆ Reads the full line including spaces. |

**Intermediate-Level Questions & Answers**

| No. | Question | Answer |
|---|---|---|
| 1 | ◆ What is widening type conversion? | ◆ Automatic conversion from smaller to larger type (e.g., int → long). |
| 2 | ◆ What is narrowing type casting? | ◆ Manual conversion from larger to smaller type (e.g., double → int). |
| 3 | ◆ Example of type casting: | ◆ double d = 45.6; int x = (int) d; → x = 45 |
| 4 | ◆ What will this print: int a = 5; System.out.println(++a + a++); | ◆ Output: 12 (pre-increment = 6, post-increment = 6, then a becomes 7) |
| 5 | ◆ Can you store a null in a primitive variable? | ◆ No. Primitives can't hold null values. Only objects (wrapper) can. |
| 6 | ◆ Difference between float and double? | ◆ float is 32-bit, double is 64-bit and more precise. |
| 7 | ◆ Why can't we use ArrayList<int> in Java? | ◆ Java generics only support objects; int is not an object. |
| 8 | ◆ How does Java convert int to Integer automatically? | ◆ Using autoboxing (Integer obj = 10;). |
| 9 | ◆ What happens if we mix nextInt() and nextLine()? | ◆ It may skip input due to leftover newline characters. |
| 10 | ◆ What's the difference between char and Character? | ◆ char is primitive, Character is wrapper class (object version). |

**Expert-Level / Tricky Questions & Answers**

| No. | Question | Answer |
|---|---|---|
| 1 | ◆ What is the memory difference between primitive and wrapper? | ◆ Primitives are lightweight and stored in stack; wrappers are objects stored in heap (with extra metadata). |
| 2 | ◆ Will this return true: Integer a = 100, b = 100; System.out.println(a == b); | ◆ Yes. Because values -128 to 127 are cached. |
| 3 | ◆ Will this return true: Integer a = 200, b = 200; System.out.println(a == b); | ◆ No. Because values above 127 are not cached, and == checks reference. Use .equals() instead. |
| 4 | ◆ Can we override intValue() method of Integer class? | ◆ No. Wrapper classes are final — they cannot be extended or overridden. |
| 5 | ◆ What will happen here? Integer a = null; int b = a; | ◆ Runtime error: NullPointerException during unboxing. |
| 6 | ◆ Can wrapper classes be used with collections? | ◆ Yes. That's their main purpose. ArrayList<Integer> works. |
| 7 | ◆ What are the default values of arrays? | ◆ Primitive arrays → 0, false, '\u0000'; Wrapper arrays → null |

| | | |
|---|---|---|
| 8 | ◆ Can we store mixed types in a wrapper class array? | ◆ No. Wrapper types are still typed, e.g., Integer[] only for Integer. |
| 9 | ◆ Is Integer immutable? | ◆ Yes. Just like String, wrappers are immutable. |
| 10 | ◆ What’s the difference between int a = new Integer(5); and int a = 5;? | |